The Role of HCI in CASE Tools Supporting Formal Methods

Connie Heitmeyer
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC 20375

Introduction

From 1988 through 1992, I led two research groups: the advanced interfaces section of NRL's Human-Computer Interaction (HCI) laboratory, which is developing advanced user interface techniques, and a software engineering group, which is designing formal methods for real-time systems. In 1989, a multidisciplinary team of HCI experts and software engineers, drawn from the two groups, began a new research task whose purpose was two-fold: to evaluate existing formal methods for representing and reasoning about a system's timing behavior and to build a prototype CASE toolset supporting the most promising methods. From the beginning, we recognized that the success of the CASE tools depended not only on powerful analysis methods but also on the quality of the toolset's user interface and its software design [4]. A high-quality user interface would allow developers to create, edit, and analyze specifications easily and effectively. A high-quality software design would enforce a clean separation between the user interface software and the software encoding the formal methods. A clean separation would facilitate software changes.

Since 1989, we have continued to design and build CASE tools supporting formal methods. In addition to our original task, which focuses on real-time methods and tools [2], we began a new, complementary task in 1991. The new task's purpose is to construct tools supporting the formal specification and analysis of requirements based on the Software Cost Reduction (SCR) approach [8, 9]. This approach, pioneered at NRL and extended recently by D. Parnas [12], emphasizes functional (rather than timing) behavior. Unlike the real-time toolset, which is based on a graphical specification language, the SCR approach relies on a tabular notation. In this paper, I provide a brief overview of the two toolset efforts, describe some HCI issues we encountered in developing the CASE tools, and conclude with a summary of open research issues involving HCI and software engineering.

Background

Real-Time Toolset. The real-time toolset, called MT, is a collection of integrated tools for specifying and analyzing real-time systems based on a graphical language called Modechart [10]. The toolset, which provides facilities for generating the graphical specifications, supports a direct-manipulation style user interface for creating, moving, and deleting specification components and several advanced features, such as zooming, undo and redo operations, an object finder (which centers the display on a named object), and a facility that improves the layout of the specifications. Users may symbolically execute the specifications with an automatic simulation tool to make sure that the specified behavior is what was intended. They may also invoke a mechanical verifier that uses model checking to determine whether the specifications imply any of a broad class of safety assertions.

An initial version of MT containing 7.8MB of run-time code is complete [2], and an extensive user guide is available [13]. The next step is to evaluate the toolset. Previous research has shown that a good way to evaluate a user interface design is to have real users use the system to perform real tasks [3]. To evaluate and demonstrate MT, we are applying it to the specification and analysis of the control system of a tactical air-launched missile that detects ground targets based on their electromagnetic emissions.

SCR Toolset. To specify software requirements, the SCR approach relies on a collection of tables. Although many methods have been proposed for formally representing functional requirements, the SCR tabular approach is one of the few that has been shown to scale up. The tabular notation provides a mathematically precise, compact approach to specifying requirements and has been applied to a wide range of real-world systems, many of them safety-critical. These latter include avionics applications, such as [9], and the software for two nuclear power plants.

Like MT, the SCR toolset will contain different kinds of tools, including a tool for generating tabular specifications, a simulator , and a verifier. In addition, the toolset will support consistency checking. In [5], we summarize the state-transition model that underlies the SCR approach to formal requirements specification. The consistency checker tests whether a set of requirements specifications satisfies the model; for example, the checker checks for type correctness, domain coverage, deterministic (rather than nondeterministic) behavior, and well-formedness of expressions describing conditions and events. Although the specification generator, the simulator, and the verifier are all still in the design stage, prototype versions of the consistency checker were built and applied to the requirements specifications of an avionics system [9]. The prototype tools proved to have considerable utility, detecting many significant errors that were overlooked by reviewers of the specifications at NRL and another Navy laboratory [5].

HCI Issues: General

A major goal in both toolset efforts is to develop user interface designs, software designs, and prototype tools that can serve as the basis for specifying and building production-quality CASE tools. In developing the two toolsets and in applying the real-time toolset to the missile application (as well as to other smaller examples [6]), we have had to wrestle with several issues concerning the design and construction of interface software. Many of these have general relevance to the role of HCI in software engineering.

Prototyping and Iterative Development. Although our interface design experts spent considerable time doing a paper design of MT's user interface, prototyping and iterative development were invaluable in developing the interface. As many others have reported (see, e.g., [3]), the principles and guidelines for developing user interfaces are still too few and immature for experts to design a high-quality interface on the first try, so prototyping and iterative development are essential. Another approach that proved valuable was to use the toolset to do real tasks. This effort identified some problems (see below) that the interface designers did not anticipate.

Separation of User Interface Software and Application Software. While separation of application and user interface software remains a worthy goal, achieving this separation is some areas can be difficult. One question that arose in our development of the real-time toolset was how to handle semantic feedback. When, for example, a user is dragging a specification component across the display, semantic feedback indicates whether the semantics permit the component to be linked to another displayed component. Separation of application and user interface software makes semantic feedback hard to implement. In the end, we opted to sacrifice separation of concerns to facilitate implementing semantic feedback.

Further, we found that the software engineers and the user interface designers were making different assumptions about what software components should be designed for ease of change. To the user interface designers, the goal of software design is to separate user interface and application code to facilitate changes in the user interface code. The goal of software design based on information hiding is to identify likely areas of change and to assign changes that will occur independently to different modules. While on the surface, these two approaches appear compatible, in practice they often are not. An implicit assumption that user interface designers make is that the user interface code (often called the presentation code) and the dialogue will change more often than the application code. A software engineer also needs to facilitate changes in parts of the software that are unrelated to the user interface.

User Interface Look and Feel. Because we are developing the SCR toolset on top of Unix, we considered several available Unix interfaces, including Motif, OpenLook, and InterViews. We selected Motif, largely because it is becoming an industry standard and thus appears more stable than the others. While its benefits outweigh its shortcoming, we have had many problems with Motif. First, Motif's widgets often fail to meet our needs. In building the SCR toolset, for example, we needed a table widget. Developing such a widget with Motif was both difficult and time-consuming. Second, the learning curve for Motif is very steep. Motif is built on the X Toolset Intrinsics (Xt) and Xt is built on X. To use Motif effectively, one needs to learn all three. We also encountered other problems: Motif widgets often hide information that a developer needs; Motif does not allow inheritance; the interaction between widget resources can be difficult to comprehend; and setting resources in an external file often produces unexpected results.

User Interface Toolkits. In many cases, implementers can simplify the construction of the user interface by using a user interface toolkit. (In this discussion, we do not distinguish between user interface toolkits and user interface development systems.) We examined a number of these, including UIM/X, XDesigner, and Builder Xcessory, all of which require Motif. An examination of the requirements of the SCR toolset revealed that most of the user interface objects we needed were unsupported by current toolkits. To construct the required objects, we would need to combine a number of the objects provided by a particular toolkit. To provide the needed interface functions, we would need to write a large portion of the interface code ourselves. Due to these shortcomings, we decided not to use any of the toolkits.

Role of Formal Methods. In my view, formal methods can have an important role in developing reliable, effective computer systems. However, based on our experience in developing CASE tools, I am skeptical that, for systems with complex interface software, formal specification of the user interface is a worthwhile (or even achievable) goal. Much of my experience in specifying systems formally has been in the area of control systems (e.g., [7]). Such systems can be represented effectively by a state-transition model that responds to an environmental stimulus (e.g., water temperature exceeds some threshold) by taking some environmental action (e.g., sounding an alarm). Developing formal descriptions of the inputs and outputs of a control system, which usually has a trivial user interface, is far easier than developing formal descriptions of a system with a complex user interface. The reason is that, in a system with a complex user interface, many different user interface designs are possible, each involving a complex set of trade-offs and one better than the next only for certain tasks. Unlike control systems, in complex user interfaces, there is no unique model of interface objects and operations that we can discover (this model is what Fred Brooks has called the "essence" of the software product).

HCI Issues: User Interfaces for CASE Tools

Some issues we confronted in developing the toolsets concern the interface design of CASE tools. We summarize two of these issues below.

Graphical vs. Tabular Notations. Graphical notations provide developers with an intuitive, easy to understand description of the system behavior. A problem with the graphical notations is that they fail to represent large specifications very well. Our solution is to allow the developers to selectively omit certain information from the visual display of the specifications. Without this approach, graphical specifications of large systems become incomprehensible. As we expected, the tabular notations allowed a large quantity of detailed information to be expressed precisely and concisely. But, unlike the graphical notation, the tabular specifications give the developer little intuition about the system being specified. Clearly, both notations serve an important purpose, but how and when to use one notation rather than another is an open question. We along with others (see, e.g., [11]) have some ideas about how the graphical and tabular notations might be combined, but more scientifically-based guidance is needed.

Creating Specifications in a Graphical Language. We discovered that, in constructing the specifications for the missile software, the developers did not use the generation tool provided by our toolset. They preferred instead to create the specifications in a textual language using an ordinary text editor. To support this textual input mode, the toolset contains a facility that automatically translates the specifications from the textual to a graphical form. The developer can then use the toolset's automatic layout program to display the result. When asked why they preferred the textual input mode, developers said that the graphical input mode was too incremental. Many small steps were needed to perform each user operation. Is this an inherent limitation of graphical specification languages or are more streamlined user commands feasible? Did the interface designers produce a user interface that is appropriate for novices but not suited to experts? As above, more scientifically-based guidance is needed.

Research Agenda

The above discussion suggests the following research agenda:

Improved User Interface Toolkits. Clearly, programs such as Motif need to be more user-friendly. Moreover, user interface toolkits need to support more functions. One possibility is to develop user interface toolkits for specific application domains. This has already been done to some extent for office systems (e.g., the Macintosh toolbox).

Principles for Developing Interfaces for CASE Tools. Research is needed that provides principles for designing effective interfaces for CASE tools. Although many HCI researchers have criticized laboratory experiments as having too narrow a focus, carefully designed laboratory experiments may shed some light on how to design user interfaces that complement human limitations and exploit human strengths. Some evidence exists that such experimentation can be fruitful. For example, in [1], four different user interfaces---one tabular, a second graphical, and two hybrids---were used to evaluate situational awareness in an avionics application. The results showed that neither the tabular approach nor the graphical approach was better for all user tasks. The graphical interface proved better than the tabular interface for doing some tasks, the tabular was better for doing others. Similar experiments are needed to determine trade-offs in the design of user interfaces for CASE tools.

Architectures for Software with Complex User Interfaces. Previous architectures for user interface software divided the software into a presentation component, a dialogue manager, and a set of application interfaces. This architecture not only has problems with semantic feedback. In addition, the dialogue component can become large and unwieldy. This is another area where domain-specific solutions should be explored. Interfaces for different application domains will likely require different software architectures.

Acknowledgments.

This paper has benefited from numerous discussions with my NRL colleagues, M. Perez, A. Rose, A. Bull, and P. Clements.

References

[1] J. Ballas, C. Heitmeyer, and M. Perez, “Aspects of Direct Manipulation in Advanced Cockpits,” *Proc., CHI ‘92 Human Factors in Computing Systems Conf.,* May, 1992, Monterey, CA.

[2] P. Clements, C. Heitmeyer, B. Labaw, and A. Rose, "MT: A Toolset for Specifying and Analyzing Real-Time Systems," *Proc., Real-Time Systems Symposium,* Raleigh, NC, Dec., 1993.

[3] J. D. Gould and C. H. Lewis, "Designing for Usability--Key Principles and What Designers Think,” *CACM* 28, 3, 1985, 300-311.

[4] C. L. Heitmeyer, P. C. Clements, B. G. Labaw, and A. K. Mok, "Engineering CASE Tools to Support Formal Methods for Real-Time Software Development," *Proc., Fifth Intern. Workshop on Computer-Aided Software Engineering,* Montreal, CAN, 6-10 July 1992.

[5] C. L. Heitmeyer and B. G. Labaw, "Consistency Checks for SCR-Style Requirements Specifications," NRL Report 9586, Wash., DC, Dec. 1993.

[6] C. L. Heitmeyer and B. G. Labaw, “Requirements Specification of Hard Real-Time Systems: Experience with a Language and a Verifier,” in *Foundations of Real-Time Computing: Formal Specifications and Methods,* A. van Tilborg and G. Koob, eds., Kluwer Academic, 1991.

[7] C. L. Heitmeyer and J. D. McLean, “Abstract Requirements Specification: A New Approach and Its Application,” *IEEE Trans. on Software Eng*., Sept., 1983.

[8] K. L. Heninger, “Specifying Software Requirements for Complex Systems: New Techniques and Their Application,” *IEEE Trans. on Software Eng.*, Vol. SE-6, January, 1980.

[9] K. L. Heninger, D. L. Parnas, J. E. Shore, and J. Kallander, “Software Requirements for the A-7E Aircraft,” NRL Rep. 3876, Wash. DC, 1978; T. A. Alspaugh and others, “Software Requirements for the A-7E Aircraft,” NRL Rep. 9194, NRL, Wash. DC, 1992 (updated version).

[10] F. Jahanian et al., "Semantics of Modechart in Real-Time Logic," *Proc., 21st Hawaii Intern. Conf. on System Sciences,* January, 1988.

[11] N. Leveson et al., “Experiences Using Statecharts for a System’s Requirements Specification,” *Proc., Intern. Workshop on Softw. Spec. and Design,* Como, Italy, Oct. 1991.

[12] D. Parnas and J. Madey, “Functional documentation for computer Systems Engineering,” Rpt. CRL 237, McMaster Univ., Hamilton, ONT, 1991.

[13] A. Rose, M. Perez, and P. Clements “Modechart Toolset User’s Guide,” NRL formal report, Wash., DC,

Dec., 1993.